TOM matching in Java

tom.loria.fr

Exercise

- Define a function

  plusInt(Nat, Nat) -> int

  which returns the addition of two Nat terms as an integer

- The instruction

  System.out.println(« res = » + `plusInt(one,two));

  should print

  ```
  res = 3
  ```

Solution (which doesn't work)

```
%gom {
  ...
  imports int
  ...
  int = plusInt(Nat, Nat)
  ...
}
module ...:rules() { ... }
```

Another solution: combine matching and Java

```
public Nat plus(Nat n1, Nat n2) {
    %match(Nat n1, Nat n2) {
        x, zero() -> { return `x; }
        x, suc(y) -> { return `suc(plus(x,y)); }
    }
}
```

- plus is no longer a defined symbol but a Java function
- x et y are variables instantiated by the matching
- they become local variables for each of the rules

Exercise

- Write the program that prints the result as a builtin integer

- System.out.println(« res = » + `entier(suc(plus(one,two))));

```
> res = 4
```

Solution

```
public int entier(Nat n) {
    %match(n) {
      zero()    -> { return 0; }
      suc(x)    -> { return 1 + entier(`x); }
    }
}
```

Right-hand side can mix Tom and Java

Semantics of %match

- Select the first pattern that matches
- Evaluate the associated action
- Execution continues
  - after the action (i.e. with the next matchable pattern) if there is no break or return
  - according to the control flow

Example

```
public int entier(Nat n) {
    %match(n) {
      zero() -> { return 0; }
      suc(zero()) -> { System.out.println("A zero!"); }
      suc(suc(zero())) -> { System.out.println("A one!"); }
      suc(y) -> { return fun(`y) + 1; }
    }
}
```

Questions

- Can we encode a conditional rewrite system with %match?
- Can we encode non-linear patterns?

Conditional rules

- Yes, the corresponding condition should be put in the action-part of the rule

```
public int entier(Nat n) {
    %match(n) {
        zero() -> { return 0; }
        suc(x) -> {
          if(`x != `zero()) { return 1 + `entier(x); }
        }
        suc(zero())  -> { return 1; }
    }
}
```

List handling

List representation

Usually we consider an empty list nil and a concatenation operator cons

$$nil : \rightarrow L$$

$$cons : E \times L \rightarrow L$$

The list a -> b -> c is represented by

cons(a,cons(b,cons(c,nil)))

Question

- How to retrieve a given element?

  in(cons(a,cons(b,cons(c,nil))),a) → true

  in(cons(a,cons(b,cons(c,nil))),b) → true

  in(cons(a,cons(b,cons(c,nil))),d) → false

Possible solution

in(cons(x,l),x) → true

in(cons(y,l),x) → in(l,x)

in(nil,x) → false

Associative symbols

- a symbol f is associative if

  f(x,f(y,z)) = f(f(x,y),z)

- which one of these statements is true?
  - f(a,f(b,f(c,d))) = f(f(a,b),f(c,d)) ?
  - σ tel que σf(x,f(c,y)) = f(f(a,b),f(c,d)) ?
  - f(x,f(c,y)) << f(f(a,b),f(c,d)) ?
  - f(x,f(c,y)) << f(a,f(b,f(c,d))) ?
  - f(x,f(d,y)) << f(a,f(b,f(c,d))) ?

Associativity with neutral element

- a symbol f is associative with neutral element e if

  f(x,f(y,z)) = f(f(x,y),z)

  f(x,e) = f(e,x) = x

- In this case do we have
  - f(x,f(d,y)) << f(a,f(b,f(c,d))) ?

Associative symbols in Tom

- A variadic symbol f with profile:

  f : E x ... x E -> L

- is denoted

  f(E*) -> L

- and we can write

  f(a(),b(),c(),d())

  f(x,a(),y,b(),z)

Associative matching

- We consider the flat forms
  - f(a,b,c,d)
  - f(x,c,y)
- and we can write
  - f(x,c,y) << f(a,b,c,d)
  - f(x,d,y) << f(a,b,c,d)

Associative symbols in Tom

```
%gom{
        module ListSet
        imports int
        abstract syntax

        Set = conc(int*)
}

public static void main(String[] args){
      Set s = `conc(1,2,3,4);
      System.out.println("s = "+s);
}
```

Question

Check if the second element of a list of four elements is a "2"

```
%match(s){
   conc(x,2,y,z) -> {System.out.println("2 found");}
}
```

Questions

- What are the types of x, y and z in conc(x,2,y,z)?
  - answer : int
- Fill in the the "..." in conc(...,2,...) to obtain a pattern that matches against conc(1,2,3,4) and conc(1,2,3)?
  - we need variables of type Set
  - which are denoted with a * like in conc(x*,2,y*)

Question

- Check if a list contains a "2"

```
%match(s){
    conc(X*,2,Y*) -> {System.out.println("2 found");}
}
```

- Check if the second element of a list is a "2"

```
%match(s){
    conc(x,2,Y*) -> {System.out.println("2 found");}
}
```

(old) Exercise

- define a signature to handle sets
- define the operations
  - occurs : E x S -> Bool
  - union : S x S -> S
  - subset, eq : S x S -> Bool
  - diff : S x S -> S

Multisets to Sets

- How to get rid of doubles?
  - conc(X*,y,y,Z*) -> conc(X*,y,Z*)
  - eliminates consecutive doubles
- How to eliminate distant doubles?
  - conc(X*,y,T*,y,Z*) -> conc(X*,y,T*,Z*)

Exercise

- Implement an addressbook with the names and birthdates of a list of persons
  - give the pattern that looks for the persons with the same name
  - give the pattern that finds the persons with a given birthday

Persons

```
%gom{
    module Persons
    imports int String
    abstract syntax

    Date = date(year:int, month:int, day:int)
    Person = person(fn:String,ln:String,bdate:Date)

    PersonList = concPerson( Person* )
}
```

Happy birthday

```
void happyBirthday(PersonList book, Date date) {
  %match(book) {
     concPerson(_*, person(fn, _, date), _*) -> {
        System.out.println("Happy birthday " + `fn);
     }
  }
}
```

Same name

```
%match(book) {
   concPerson(_*, p1@person(prenom,name,_), _*,
                  p2@person(firstname,name,_), _*)
    -> {
         System.out.println(`p1+" and " +`p2 +
                  "have the same family name");
   }
}
```

Exercise

- Given an order on the elements of the list
- Define a sorting algorithm for the list
- Implement it in Tom

```
public static PersonList sort(PersonList book) {
%match(book) {
    concPerson(X*, p1@person(prenom,nom,_),  Y*,
                   p2@person(firstname,name,_), Z*)
   -> {
    if(`nom.compareTo(`name) > 0){
      return  sort(`concPerson(X*, Y*, p2, p1, Z*));
    }
  }
}
return book;
}
```

Exercise

- Find the members of the same family that have a different firstname

Different name

```
%match(book) {
    concPerson(_*, p1@person(prenom,name,_), _*,
                   p2@person(fname,name,_), _*)
-> {
        if(`fname != `prenom)
          System.out.println(`p1+" and " +`p2 +
              "have different firstnames");
    }
}
```

Different name

```
%match(book) {
    concPerson(_*, p1@person(prenom,name,_), _*,
                   p2@person(!prenom,name,_), _*)
-> {
      System.out.println(`p1+" and " +`p2 +
              "have different firstnames");
   }
}
```

Anti-patterns

- complements used in patterns
- specify what you don't want to match
- use the '!' symbol
- !a => everything that is not a
  - g(!a) << g(b)
  - f(x,!x) << f(a,b)

Cars

```
Car = car(m:Brand,c:Color,t:Color)
    | suv(m:Brand,c:Color,t:Color)

Color = red()
      | blue()
      | green()

Brand = Ford()
      | Renault()
      | BMW()
```

Exercise

- Check if a car is
  - not a SUV
  - not red
  - with different colors in and out

Cars

```
%match(c) {
    !suv(_,_,_)  -> {
            System.out.println("; not a SUV");
    }
    car(_,!red(),_)  -> {
            System.out.println("; not red");
    }
    car(_,x,!x)  -> {
            System.out.println("; different colors");
    }
}
```

Associative lists

- list(*,x,*,x,*) : 2 identical elements
- list(*,x,*,!x,*) : 2 different elements
- !list(*,x,*,x,*) : all different
- !list(*,x,*,!x,*) : all the same

What do we have ?

- efficient data-structures (maximal sharing)
- rewrite rules (labeled and unlabeled)
- ∅, A, and AU matching (non-linear)
- anti-patterns: !conc(_*,a(),_*)
- everything, smoothly integrated into Java

Hooks

Gom

- define a data structure and
- rewrite rules
- which specify the prefered (normal) forms like
  - sorted lists
  - reduced fractions

Actions executed at construction time (hook)

```
module Bool
abstract syntax
Bool = True()
     | False()
     | Not(b:Bool)
     | And(l:Bool,r:Bool)
     | Or(l:Bool,r:Bool)
```

```
module Bool:rules() { ... }

sort Bool:interface() { ComparableTo<Bool> }
sort Bool:block() { ... }

Not:make(x) { ...}
```

Rules

```
module Bool
abstract syntax
Bool = True()
     | False()
     | Not(b:Bool)
     | And(l:Bool,r:Bool)
     | Or(l:Bool,r:Bool)
```

```
module Bool:rules() {
                Or(True(),_) -> True()
                Or(_,True()) -> True()
}

public static void main(String[] args){
        System.out.println(`Or(False(),True()));
}
>True()
```

Actions executed at construction time (hook)

```
module Bool
abstract syntax
Bool = True()
     | False()
     | Not(b:Bool)
     | And(l:Bool,r:Bool)
     | Or(l:Bool,r:Bool)
```

```
Not:make(b) {
    %match(Bool b) {
        Not(x)   -> { return `x; }
        And(l,r) -> { return `Or(Not(l),Not(r)); }
        Or(l,r)  -> { return `And(Not(l),Not(r)); }
    }
}
```

```
System.out.println(`Not(Or(False(),False())))
>And(True(),True())
System.out.println(`Not(And(False(),True())));
>True()
```

Blocks of source code

```
module Bool
abstract syntax
Bool = True()
     | False()
     | Not(b:Bool)
     | And(l:Bool,r:Bool)
     | Or(l:Bool,r:Bool)
```

```
sort Bool:interface(){ Boolable }
sort Bool:block(){
  public boolean toBool(){
    %match(this){
      False() -> { return false; }
      True() -> {  return true; }
      Neg(a) -> { return ! `a.toBool() ;    }
      Or(a,b) -> {return `a.toBool() || `b.toBool();}
      And(a,b) -> {return `a.toBool() && `b.toBool();}
    }
    return false;
}
```

Exercise

- Define the type of sorted lists

Sorted lists

```
%gom {
    module sortedList
    imports int
    abstract syntax

    Integers = sorted(int*)

    sorted:make_insert(e,l) {
            %match(l) {
                sorted(head,tail*) -> {
                    if(e >= `head) {
                            return `realMake(head,sorted(e,tail*));
                    }
                }
            }
    }
}
```